Memory Compression Techniques for Network Address Management in MPI

Yanfei Guo*, Charles J. Archer†, Michael Blocksome†, Scott Parker‡, Wesley Bland†, Ken Raffenetti† and Pavan Balaji*
*Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
†Intel Corporation, Santa Clara, CA, USA
‡Argonne Leadership Computing Facility, Argonne National Laboratory, Lemont, IL, USA

***Abstract*—MPI allows applications to treat processes as a logical collection of integer ranks for each MPI communicator, while internally translating these logical ranks into actual network addresses. In current MPI implementations the management and lookup of such network addresses use memory sizes that are proportional to the number of processes in each communicator. In this paper, we propose a new mechanism, called AV-Rankmap, for managing such logical addressing. AV-Rankmap takes advantage of logical patterns in rank mapping that most applications naturally tend to have, and it exploits the fact some aspects of the network address translation are naturally more performance critical than others. It uses this information to compress the network address management structures. We demonstrate that AV-Rankmap can achieve similar or better performance compared with that of other MPI implementations while using significantly less memory.**

I. Introduction

MPI is the most commonly used programming model for scientific computing on large supercomputing systems. Consequently, keeping up with the growing scale of such systems is a critical aspect in the design of MPI implementations. In the past few years, tremendous improvements have been made in MPI implementations with respect to avoiding data structures that scale linearly or superlinearly with system size [8]. Yet despite these improvements, MPI implementations today still have scalability limitations. One example is MPI's management of process logical network addressing.

MPI processes are logically represented as integer *ranks* within communicators, while the MPI implementation internally maintains the physical network addresses of the different processes. When a user application moves data between processes by addressing them using these ranks, the MPI implementation internally translates such ranks into the corresponding network addresses before communication can be performed. Such network address management has two closely related aspects that need to be considered. First, the network physical addresses themselves and their associated data structures need to be maintained: we refer to these as "virtual connections" or "VCs." Second, a mapping between the logical ranks to the network physical addresses needs to be maintained for each communicator: we refer to this mapping as "virtual connection reference tables" or "VCRTs."

Such structures exist in practically every MPI implementation today, even though the terminology used is sometimes different. For example, MPI implementations such as MPICH, MVAPICH, Intel MPI, Cray MPI, IBM Blue Gene MPI, Microsoft MPI, Tianhe MPI, and Sunway MPI, use the terminology of VCs and VCRTs. Open MPI and Fujitsu MPI, on the other hand, use the terminology of "proclist" and "plist", though conceptually they are no different from VCs and VCRTs.

Most MPI implementations manage VCs and VCRTs in a way that is optimized for performance. For example, the number of dereferences or lookups are minimized, and few or no branches occur in the performance critical path. Unfortunately, not much attention has been paid to the memory usage of these data structures. For example, data structures related to different transports (e.g., network or shared memory) are embedded into the VC, rather than referenced from it. Similarly, data structures related to the shared-memory topology, which are required for transport selection, are embedded into the VC for fast lookup. Such a model, while optimized for performance, costs substantial memory.

This situation is also true for the VCRTs that manage the mapping of logical process ranks to network physical addresses. Maintaining the process mapping metadata is complicated by the fact that the MPI standard allows users to create new communicators by arbitrary reordering the ranks compared to the parent communicator. That is, a single process can have two completely different ranks within two different communicators, with no correlation between the two. Because there are $P!$ (P factorial) valid mappings of communicator ranks to network addresses, where P is the number of processes in the communicator, lookup tables—such as VCRTs—remain the common practice for maintaining this metadata. Although this simple approach works for any possible reordering of ranks, it is not memory efficient. It takes $O(P)$ memory space on each process for each communicator, that is, $O(nP)$ total memory space on each process, where n is the number of communicators created by the process. Consequently, such metadata can result in the MPI implementation consuming a significant fraction of the available memory, particularly for very large supercomputers.

Our objective is to reduce the memory consumption of such network address management for MPI processes by using appropriate compression techniques. We observe that although the network address management used by current MPI implementations is the most generic model that works for any possible reordering of ranks, most applications do not need or use such generality. For example, the rank mappings used by most applications are not arbitrary: they follow a

simple, predefined pattern. Applications such as Nek5000 [5] tend to create simple duplicates of existing communicators; in such cases, the rank mapping of the new communicator is exactly the same as the rank mapping of the original parent communicator. Similarly, applications such as NWChem [18] and QBOX [6] tend to lay the processes in a virtual two-dimensional grid and split processes as "row" and "column" communicators. The "row" communicator's rank mapping then is a subset of the parent communicator's rank mapping at a fixed offset; the "column" communicator's rank mapping is a simple strided lookup based on the parent communicator's rank mapping. For such applications, the new communicator does not need a completely independent VCRT but, rather, just some simple additional information, such as the split offset or stride, and a reference to the parent communicator's VCRT in order to calculate its own mapping. Such compressed format significantly reduces the memory footprint of the required metadata, but it comes at the cost of additional computation to perform the required translation.

Based on these observations, we propose a new mechanism, called AV-Rankmap, for network address management. AV-Rankmap uses several compression techniques to minimize the memory space used for network address management. We study the behavior of AV-Rankmap with respect to two properties: memory usage and performance.

With respect to memory usage, we present in Section III a survey of various HPC applications and demonstrate that although AV-Rankmap does not improve memory usage in the worst-case scenario, it does significantly improve the common cases in which most applications fall. For example, we decouple the network address infrastructure to distinguish elements in the VC structure based on various properties such as commonality of use, compressibility, and network-specific attributes. This decoupling allows applications that use only a subset of the features in MPI, without relying on the full generality of MPI, to benefit from a smaller overall memory footprint. Similarly, for applications that create communicators with certain kinds of patterns—three forms of which are studied in this paper: *direct*, *offset*, and *strided*—we detect such patterns and use them to reduce the metadata storage. When we are unable to detect any pattern in the rank mapping, we simply fall back on the original lookup table–based model.

With respect to performance, we study both the communicator creation path (which is typically not performance critical) and the data communication path (which is typically performance critical) and measure the overhead added in each case. For the noncritical path, we aim to keep the additional cost low, although some extra overhead is often tolerable. The additional cost is due primarily to the rank-mapping pattern detection involved in the AV-Rankmap approach. We use simple pattern detection techniques, although one could use more sophisticated techniques that might have higher overhead.

For the performance-critical path, however, the overhead needs to be practically negligible for the proposed approach to be viable. For most MPI implementations performance is the most significant metric for measuring impact and the general motto for incorporating new techniques is: if the performance overhead of the proposed approach is not zero, it is too high! For a more quantitative measure, a high-performance and finely-tuned MPI implementation would cost as low as approximately 50 instructions on the communication path all the way from the application to the low-level network communication layer (e.g., in `MPI_Put`). Adding a single additional instruction in this path would lead to a 2% overhead in performance (assuming, for simplicity, that all instructions are equally expensive). Keeping this in mind, we perform a detailed instruction and cache-level analysis of the performance-critical path and illustrate that although the AV-Rankmap approach adds a few additional instructions for address translation in the performance-critical path, the lost performance due to the additional instructions is more than compensated by the improved cache activity because of the smaller metadata footprint, thus leading to similar or better performance while using significantly less memory.

The overall design of AV-Rankmap and a detailed evaluation with both microbenchmarks and real applications are showcased in this paper on up to 786,432 MPI processes. We demonstrate that AV-Rankmap can improve the memory usage of current MPI implementations by several orders of magnitude in many important cases.

II. Communicator Creation in MPI

At initialization time, MPI implementations create two communicators by default: `MPI_COMM_WORLD` and `MPI_COMM_SELF`. After initialization, applications are allowed to create additional communicators dynamically. A new (child) communicator is typically created from one existing (parent) communicator, although routines also exist that allow processes from two existing parent communicators to be combined into a single communicator.

Communicator creation routines can be broadly classified into four categories.

Duplicate Communicators. The most common communicator creation model used in MPI is duplication. MPI provides three routines in this model: `MPI_Comm_dup`, `MPI_Comm_idup`, and `MPI_Comm_dup_with_info`. These routines allow the application to create a new communicator with a rank mapping identical to that of the parent communicator but with a different communication context.

Split Communicators. The second communicator creation model is communicator splitting. Routines such as `MPI_Comm_split` fall into this category, in which the application can split the available processes into multiple disjoint groups (using a unique *color* to identify each group), where each group forms its own new communicator. In this model, the application can reorder the ranks of the processes in the new communicator, potentially in an arbitrary fashion. In practice, however, most applications do not reorder processes arbitrarily (or at all) when creating such split communicators. Thus, the resulting child communicators are often simple subsets of the parent communicator, with a well-structured mapping between the ranks of each process on the parent and child communicators. Other routines such

as MPI_Comm_create, MPI_Comm_create_group, and MPI_Comm_split_type also fall in this category.

Topology-Aware Communicators. Topology-aware communicator creation routines such as MPI_Cart_create, MPI_Dist_graph_create and MPI_Graph_create allow applications to map application communication topologies to physical hardware topologies. For all these routines, two forms exist: with and without reordering. Reordering allows the ranks of the child communicator to be reordered in a topology-aware fashion (to allow for faster communication between some ranks compared with others). When reordering is disabled, the child communicator has a rank mapping identical to that of the parent communicator. In practice, however, topology-aware communicators are not widely used. In the cases where they are used, reordering is typically not applied. And in even when reordering is applied, most current MPI implementations do not perform any actual reordering of processes, with a few notable exceptions such as MPI_Cart_create on Blue Gene supercomputers.[1] The outcome is that, in the vast majority of cases, the parent and child communicators have identical rank mappings. In cases where reordering is applied, the mapping is not identical, but is computable based on the hardware topology, though we do not do so in this paper.

Intercommunicators. Intercommunicators are created by using MPI_Intercomm_create, by dynamically spawning additional MPI processes (e.g., MPI_Comm_spawn), or by connecting multiple MPI applications (e.g., MPI_Comm_connect). An intercommunicator has two nonoverlapping groups: a remote group and a local group. Rank translations for these two groups are handled separately. Intercommunicators are rarely used in large applications. Most applications tend to rely on traditional communicators (called "intracommunicators") created by using a routine from one of the other three categories. The reason is threefold. First, typical application use cases do not map well to intercommunicators where communication is not as intuitive (e.g., a process can communicate with processes only in the remote group and not its local group). Second, not all routines in MPI are "intercommunicator-safe"; for example, one-sided communication windows are undefined over intercommunicators. Third, large supercomputers such as the IBM Blue Gene and Cray XE, XK, and XC systems do not support creating intercommunicators. As application workflows that allow multiple applications to connect to and communicate with each other gain more traction, we expect intercommunicators to be more widely used. Nevertheless, intercommunicators are unlikely to ever become the primary communication model for most applications.

An additional aspect to note here concerns internal communicators. In most MPI implementations, for every communicator that is created by the user, the MPI implementation internally creates additional convenience communicators. A common example is convenience communicators created for topology-aware collectives. For example, for each user-created communicator, some MPI implementations create one additional internal subcommunicator containing processes that reside on the same node and a second internal subcommunicator containing one root process from each node. Thus, in essence, for each user-created communicator a total of three communicators is created, although not all of them contain the same number of processes.

[1]This is expected to change in the future: many MPI implementations are actively working on improving topology-aware communicator creation, and reordering ranks when needed.

III. A Survey of Communicators in Applications

Section II described the different communicator creation techniques available in MPI. In practice, however, some techniques are more commonly used than others. To understand the communication creation models used in various applications, we performed a survey of a large number of applications comprising the NAS parallel benchmarks [4], CORAL benchmarks [3], DOE codesign applications [1], [2], and other large applications that consume significant compute cycles on large supercomputing centers [18], [5]. Although our survey covered 62 different applications, we highlight only a small subset of the survey here, because of space constraints.

Nek5000. Nek5000 is a highly scalable spectral-element method for solving computational fluid dynamics problems. Its computational model relies on solving computational grids at an increasing level of refinement. That is, for a given computational problem, it creates multiple grids, each at a different granularity or coarseness. Solving each grid gives an approximate solution to the problem. Solving the next-finer grid then refines the result based on the approximation generated by solving the previous coarser grid.

Because each grid requires its own communication context, Nek5000 creates a new communicator for each grid. These communicator are essentially duplicates of MPI_COMM_WORLD, and the number of communicators created increases as the number of levels of refinement desired by the application increases. A rule of thumb is that as the problem size grows, the number of refinement levels (and hence the number of communicators created) grows as well. In current production runs of the application, a "medium-scale" problem typically creates 24 refinement levels (i.e., 24 new communicators), and a "large-scale" problem typically creates 86 refinement levels (i.e., 86 new communicators). Medium-scale problems are considered appropriate to scale up to 16–32K processes, while large-scale problems are considered appropriate to scale up to the full scale of the current largest supercomputers in the world.

NWChem. NWChem is a quantum chemistry application suite featuring a broad set of simulation capabilities targeted at many areas including quantum simulation of molecules with heavy isotopes and multiscale methods for modeling aqueous chemical reactions relevant to environmental chemistry. For its core linear algebra computations, NWChem (using libraries such as ScaLAPACK) creates virtual two-dimensional data grids on which the computation is carried out. To improve scalability, it splits the MPI processes into "row" and "column" communicators using MPI_Comm_split. Each process is a part of a "row" and a "column" communicator in this model.

Data exchange and synchronization are then limited along these smaller communicators, thus improving performance and scalability. We note that during the split, process ranks are not reordered. Thus, the ranks of the processes in the "row" communicator are essentially the same as the ranks of the processes in `MPI_COMM_WORLD` but are offset by a constant value. Similarly, the ranks of the processes in the "column" communicator can be calculated with fixed offset and stride values.

Apart from the core linear algebra computations, NWChem spends a large fraction of its computation on force calculations. These are typically done through one-sided communication that, after passing through multiple layers of the software stack, eventually uses MPI-3 one-sided communication (or RMA) windows internally. For each RMA window, the MPI implementation internally creates a new communicator that is a duplicate of the parent communicator from which the RMA window is being created, in order to perform the required data movement and synchronization. NWChem itself creates three to four RMA windows, depending on the problem being solved. However, when used together with the Casper [16] software stack for asynchronous progress, for each window created by NWChem, Casper creates as many duplicate windows as the number of cores on each node of the machine. For example, when NWChem is executed on the IBM Blue Gene/Q with 16 processes on each node, it creates 64 RMA windows (and thus 64 new communicators). When it is executed on the Intel Xeon Phi Knights Landing with 60–70 processes on each node, it creates on the order of 300 RMA windows (and thus 300 new communicators).

HACC. HACC is an astrophysics framework that simulates the formation of structure in the expanding universe. HACC's computational model is similar to the linear algebra portion of NWChem in that they both rely on multidimensional data grids for their computation. However, HACC does not split its communicators; instead, it creates multiple topology-aware Cartesian communicators (using `MPI_Cart_create`) representing three-dimensional, two-dimensional, and one-dimensional distributions of the problem. Like NWChem, HACC does not reorder the processes in the new communicator. Thus, apart from the additional topology information that is attached to the communicator, the process mapping itself is identical to that of the parent communicator.

Summary. Table I summarizes the communicator creation models used in a number of applications. In this table, we have classified the communicator creation models into several categories. The "dup" column refers to the cases where a duplicate communicator is created either directly by using one of the communicator duplication functions or indirectly by, for example, creating an RMA window on a communicator. The "split" columns refer to the cases where a communicator is split into smaller subcommunicators. "Offset" refers to the case where the process ranks in the new communicator are identical to that of the parent communicator, but at a fixed offset; "stride" refers to the case where the process ranks in the new communicator can be calculated based on a fixed offset and stride from the parent communicator; and "irreg" refers to the case where no pattern in the mapping is detected. The "topo" column refers to the cases where a topology-aware communicator is created. The "intercomm" column refers to the case where dynamic spawned or connected processes are used.

TABLE I: Mapping models for communicators.

Application	Dup	MPI_Comm_split			Topo	Intercomm
		Offset	Stride	Irreg.		
Nek5000	x					
QMCPACK		x	x			
NWChem	x	x	x			x
HACC					x	
QBOX	x	x				
CAM-SE	x	x				
NAMD	x	x				
LSMS		x		x		
SP	x	x				
BT	x	x	x			
FT	x	x				
Graph500	x	x	x			
Nekbone	x					
SNAP	x				x	
MCB	x					
cian2		x				
MCCK		x	x			
mocfe_bone		x	x			
pynamic	x	x	x			
MACSio	x					
AMG2013	x					
CNS	x					
SMC	x					
AMR	x					

Based on the summarized information in Table I, we note a few important points.

1. The most commonly used communicator creation model is that of communicator duplication (either by using `MPI_Comm_dup` or by creating RMA windows). While the duplication itself is straightforward, as noted in Section II, the MPI implementation creates multiple internal communicators for each user communicator. These internal communicators, however, are not simple duplicates of the parent communicator. Thus, in some sense, no "pure" duplication of communicators is possible in modern MPI implementations.

2. Split communicators (particularly, offset based and stride based, such as those described for NWChem) are heavily used. The offset-based model is the more common model and is valuable in splitting a multidimensional process grid of any number of dimensions. A common scenario where the offset-model of splitting is used is when the application wants to divide the available processes into smaller groups of fixed sizes. The stride-based model, on the other hand, is valuable in splitting a multidimensional process grid when the number of dimensions is larger than one.

3. Irregular splitting of communicators occurred in a single application, LSMS. We note, however, that here "irregular" does not mean that the application does not have any pattern in splitting the communicator. In fact, a real application is unlikely to create a split communicator with no pattern whatsoever. Here "irregular" means only that the pattern used by the application does not fall into the fixed set of patterns that we automatically detect.

4. Topology-aware communicators are used by two applica-

tions: HACC and SNAP [7]. Both use `MPI_Cart_create` to create these communicators, and neither application reorders the process ranks. Thus, the topology-aware communicators in these cases are essentially regular duplicate communicators, as far as process mapping is concerned.

5. Intercommunicators are rarely used and were observed only in NWChem. However, the use of intercommunicators in NWChem is not for creating dynamic processes; rather, it is a mechanism for creating regular intracommunicators, but in a noncollective fashion. MPI-2 did not provide functionality to create communicators in a noncollective fashion, and hence researchers attempted to do so by using intercommunicators, as described in [10]. In fact, the intercommunicators created in this process are temporary and are freed as soon as the final intracommunicator is created. MPI-3 included this functionality in `MPI_Comm_create_group`, which is simpler and more efficient compared with the method used by NWChem. At the time of our writing this paper, however, NWChem has not yet been updated to use this new MPI-3 functionality. When it does move to this new MPI-3 functionality, it will no longer have a dependency on intercommunicator creation. So, in a way, this dependency is a temporary artifact of the current implementation of NWChem.

IV. Design of AV-Rankmap

As described earlier, network address management has two closely related aspects that need to be considered: (1) network physical addresses themselves and their associated data structures (i.e., VCs), and (2) a mapping between the logical ranks to the network physical addresses (i.e., VCRTs). The AV-Rankmap model retains the concept of VCs and VCRTs but optionally adds one additional level of abstraction, as we describe in this section.

First we focus on the traditional VCRT-VC model, the data structures it maintains and the advantages of such a model. Different MPI implementations use different terminologies for these structures, but these concepts exist in almost every MPI implementation available. Then we describe the AV-Rankmap model, including the overall design of the proposed approach, how it differs from the VCRT-VC model, and its benefits and disadvantages compared with the VCRT-VC model.

A. Traditional VCRT-VC Model

The traditional VCRT-VC model used in most MPI implementations uses a simple two-level hierarchy. At the top level is a VCRT structure, which essentially is a collection of pointers to each VC structure. The VCRT is an $O(P)$ structure that is statically allocated at initialization time.

At the bottom level is a VC structure that contains the required information for communicating with a process. The VCs themselves can be fully dynamically allocated, in theory, for example at the time of the first communication with the corresponding process. However, most MPI implementations today choose to statically allocate a "small part" of the VC structure (basic bookkeeping information) and dynamically allocate the more expensive portions of the VC on demand (such as network connections and communication buffers). Thus, the VC structures result in another $O(P)$ memory space.

The VC structure is organized to minimize the number of dereferences. Consequently, all the information required for communication in embedded into this structure, rather than referenced from it. We classify the elements of the VC structure into three categories: core network access information, multitransport functionality, and functionality for dynamic processes.

Core Network Access Information. The core network access information refers to the basic network-specific functionality that is necessary for accessing a remote process. This includes information such as target endpoint information. For example, for InfiniBand, this would be the queue-pair information to which we can send data. Such information is the most basic and essential part of the VC and is required for any communication operation.

Multitransport Functionality. Almost every MPI implementation allows for data to be communicated over multiple transports. At least two transports are provided by all MPI implementations—shared memory for intranode communication and a network interconnect for internode communication—although some MPI implementations allow for more than two transports to be used simultaneously. Each transport has its own collection of information, such as communication functionality to use, communication thresholds for eager/rendezvous communication and queues for temporary communication buffers. For fast lookup, such information is directly embedded into the VC structure itself, thus avoiding a dereference. The cost of doing so, however, is that (1) the transport-specific information is replicated a large number of times across the different VCs and (2) some VCs might maintain more information than what they need for communication with the peer process that they correspond to (despite minimizing such additional information using unions).

Functionality for Dynamic Processes. Dynamically spawned or connected processes is a core part of the MPI standard, although it is rarely used in applications. However, current VC structures tend to give such functionality importance (with respect to performance) equal to that of more commonly used functionality such as basic send/receive communication. Consequently, elements that are required to implement dynamic processes are embedded into the VC structure as well and use up memory space irrespective of whether the application uses dynamic processes or not.

B. AV-Rankmap: Compressing the VC

The AV-Rankmap model aims at improving the traditional VCRT-VC model in two ways: (1) reducing the size of the VC structure based on various properties such as commonality of use, compressibility, and network-specific attributes and (2) reducing the $O(P)$ memory usage of the VCRT structure, where possible, by detecting rank mapping patterns.

As described in Section IV-A, the traditional VC structure has three classes of components. Of these, the core network access information is the most critical part.

Compressing the Multitransport Functionality. The multitransport functionality in the traditional VC structure is highly

redundant since the number of transports used is typically much smaller than the number of VCs. The number of VCs is equal to the total number of processes in the system, while the number of transports is equal to the number of networks being used (which is typically just two: shared memory and an internode network). Consequently, if we can decouple the transport-specific functionality from the VC, such information can be highly compressible. The challenge, however, is that such decoupling needs to be done in a way that it still retains fast lookup of this data, which was the primary reason these fields were embedded into the VC in the traditional model. We need to consider two sets of transport-specific variables: (1) a single variable that identifies which transport to use and (2) a collection of variables that are used by the transport itself. These two sets have very different properties.

Identifying Which Transport to Use. The variable that identifies which transport must be used for a given peer process should either be embedded directly into the VC structure (as in the traditional model) or be easily and quickly computable. Fast computability is, unfortunately, not easy particularly when the processes are not laid out in a homogeneous manner. Thus, we chose to always store this information inside the VC structure, using just enough bits to store the number of available transports (single bit for two transports). While, in theory, this is an $O(P)$ data structure, in practice, network transport addresses tend to have unused bits that can be used here without adding additional memory overhead. For example, the libfabric network API allows network transports to use 63-bit network addressing, thus leaving behind one bit for such transport-selection functionality. Similarly, the UCX network API uses aligned pointers for network addressing where the last 2-3 bits are unused (depending on whether the alignment is 4-byte or 8-byte). Extracting this information at runtime requires a bit-mask or bit-shift operation, which is a single (fast) instruction on most architectures.

Accessing Transport-specific Information. Decoupling transport-specific information from the VC structure improves compressability, but it will add an additional address dereference (e.g., a pointer lookup) to access this information. There is no escaping this dereference, unfortunately. But we can attempt to minimize its cost.

There are two costs associated with this additional derefenrece: (a) cache penalty for looking up the additional information, and (b) instruction costs (or instructions per cycle). Of these, based on our analysis, we expect the cache penalty to not be a significant issue. Specifically, for applications that perform frequent communication, these fields would already be in the processor cache anyway, and embedding them inside the VC structure or not does not add any additional penalty as long as the processor cache has sufficient associativity. On the other hand, for applications that do not perform frequent communication and might not be able to retain the transport-specific information in their cache, the communication cost itself would likely not be as big a concern and the additional cache-miss to access this information would likely not be as important.

For the instruction costs, however, we have not yet been able to identify a satisfactory solution. The additional dereference results either in additional instructions (to load the transport-specific information to registers) or in more expensive instructions (e.g., memory-based instructions rather than register-based instructions). Even if the data is in cache, memory-based instructions seem to be fairly expensive compared to register-based instructions thus impacting the instructions per cycle that we can achieve. While this is certainly a concern, as we will demonstrate later, the smaller memory footprint of the AV-Rankmap approach leads to fewer cache misses, which more than compensates for such additional instruction cost. Thus, from an overall performance perspective, such decoupling of transport-specific information is still a win.

Deprioritizing Dynamic Processes. As described in Section IV-A, the traditional VC structure gives equal importance to all fields, irrespective of how widely they are used in applications. In particular, dynamically spawned or connected processes need additional information such as which process group they belong to. Embedding this information into the VC can improve performance, but it also increases the size of the data structure for applications that do not use them. In the AV-Rankmap model, we deprioritize dynamic processes such that applications that do not use dynamic processes use lesser memory. However, this deprioritization comes at a cost: applications that do use dynamic processes can, in some cases, use more memory than the traditional VCRT-VC model.

Specifically, communicators that contain a combination of processes such that some of them are from one `MPI_COMM_WORLD` and some others from a different `MPI_COMM_WORLD` (i.e., dynamically spawned or connected processes) need to maintain two pieces of information: which process group does the remote process belong to and what is its rank within that process group. In the traditional VCRT-VC model, both these pieces of information were stored inside the VC structure. In the AV-Rankmap model, however, we move this information out of the VC and into the communicator structure. The benefit of this model is that for applications that do not have such communicators with processes that belong to multiple `MPI_COMM_WORLD`s, no additional memory is used. However, consider an application that creates multiple communicators containing various collections of dynamically spawned processes: for such applications, each such communicator would need to store this additional information, thus costing more memory than the traditional VCRT-VC approach does.

Fortunately, as we describe in Section IV-C, this is the worst-case scenario. In most cases, we can detect patterns in the formation of such communicators and can substantially compress this information.

C. AV-Rankmap: Rank-Address Translation

Rank-address translation is essentially the process of finding the appropriate network address to communicate with given a communicator and a rank within that communicator. Here we first describe the general model in which such communication information is stored in the AV-Rankmap model. We then describe how the optimizations utilize patterns in the rank mapping to compress the amount of memory used.

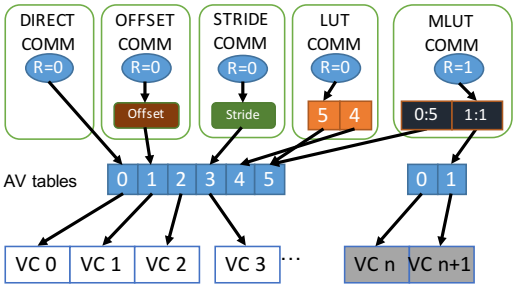


Fig. 1: Process address translation scheme.

In the AV-Rankmap model, when the application initializes, each process creates an address vector (AV) that stores two pieces of information for each target process: (1) either the VCs themselves or a pointer to the VC, and (2) which communication transport to use for that target process. For some networks, the size of the network address is at most 64 bits, so it is intuitive for the VC to be directly embedded into the AV, thus avoiding an additional dereference. For any given process, the index within the AV containing the VC for that process is referred to as the *lpid* (local process ID). If the application spawns or connects to another `MPI_COMM_WORLD`, then a new AV is created that stores the VCs corresponding to the new group of processes. Each such spawned or connected group is referred to as a "process group" and has its own unique ID, called the *pgid*. A *pgid* refers to both the process group and the AV associated with that process group. Thus, a *pgid* and an *lpid* together can uniquely identify any process in the application. The rank-address translation process can be formally represented as the following mapping.

$< comm, rank > \rightarrow < pgid, lpid >$

Once the AV and the corresponding VCs are created, the next step is to create a mapping between the communicator and one or more AVs. This is illustrated in Figure 1. As discussed in Section III, for the communicators created in most applications, the rank mappings are not arbitrary: they follow a simple, predefined pattern. AV-Rankmap takes advantage of this behavior to try to identify common patterns and use this information to compress the memory space required for maintaining such mapping. It identifies three regular mapping models: *direct*, *offset*, and *stride*, which represent the most common use cases in applications.

The ***direct*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order as `MPI_COMM_WORLD`. In this model, we do not need any additional storage other than the AV. The index in the AV (i.e., the *lpid*) is the same as the communicator rank in this model. Communicators that are duplicates of `MPI_COMM_WORLD` fall into this model. We note that in the traditional VCRT-VC model, such communicators still needs at least one $O(P)$ VCRT.

The ***offset*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order, but at a fixed offset, as `MPI_COMM_WORLD`. In this model, apart from the AV itself, the only additional piece of information that needs to be stored is the offset. The index in the AV (i.e., the *lpid*) can be calculated as the communicator rank plus the offset, in this model. Communicators that have been split without reordering from `MPI_COMM_WORLD` or one of its duplicates fall into this model.

The ***stride*** model allows the rank in a communicator to be mapped to a noncontiguous subgroup of `MPI_COMM_WORLD` with a fixed stride. The stride model has three parameters: stride, blocksize, and offset. The stride is the interval between the start of each block. The blocksize is the number of processes in each block. The offset is the offset of the rank 0 of the communicator.

All three regular mapping models use constant memory regardless of the number of ranks in the communicator. We note that these regular mapping models are valid only for communicators where all ranks are in the same process group, that is, there are no dynamic processes ($pgid = 0$).

When the rank-process mapping does not fit any of the regular mapping models, AV-Rankmap falls back on irregular mapping models using a rank lookup table. We designed two lookup tables for irregular mapping: *lut* and *mlut*. The lut is a dense array of lpids. It is used when all the ranks are in the same process group. The mlut is an array of $< pgid, lpid >$ pairs. It is used only when some of the ranks in the communicator belong to a different process group from others. We note that the lut requires only that all ranks have the same pgid; this pgid does not need to be zero. An example is an intercommunicator that is created when a new process group is connected. The remote group and the local group have different pgids, but the ranks in each group have the same pgid. Hence, both the remote group and the local group are represented by using lut instead of mlut. We need mlut only when we merge these two groups together using `MPI_Intercomm_merge`.

Note that there is an inclusive property in both the regular and the irregular mapping models. For example, a direct model can also be described as an offset model with the offset value equal to zero. This inclusive property is carefully exploited during communicator creation.

1) Creating the Rank-Address Translation for a Communicator: Users create new communicators based on existing communicators. When a new communicator is created, we have two pieces of information: (1) the mapping array that maps the ranks from the new communicator to the ranks in the parent communicator and (2) the compressed mapping model that allows us to translate a rank in the parent communicator to the corresponding AVs and lpids. Our task here is to create a similar compressed mapping model that allows us to translate a rank in the new communicator to the corresponding AVs and lpids. This is done in three steps. First, we try to detect a pattern in how the ranks in the child communicator correspond to the ranks in the parent communicators. Second, we use this detected pattern together with any mapping pattern that exists between the parent communicator and the AVs/lpids, to generate a new mapping pattern between the child communicator and the AVs/lpids. Third, if there are multiple parent communicators and the child has an irregular mapping, we try reduce it to one of the regular mapping models.

During the creation of a communicator, the MPI imple-

TABLE II: Child communicator mapping models for a given parent communicator rank map (row) and indirect mapping model (column).

	Direct	Offset	Stride	Irregular
direct	direct	offset	stride	lut
offset	offset	offset	stride	lut
stride	stride	lut	lut	lut
lut	lut*	lut*	lut	lut
mlut	mlut*	mlut*	mlut	mlut

mentation first creates an *indirect mapping array* between the ranks in the child and the parent communicators. This information directly follows from the communicator creation function being used. Once this information is obtained, the next step is to convert this indirect mapping array to a compressed mapping pattern. To this end, we assume that the mapping pattern is offset based, and we calculate the offset value based on *rank 0* in the child communicator. We then try to validate the offset value with the remaining ranks in the indirect mapping. If successful, we set the mapping pattern between the child and parent communicator ranks to be offset-based. If not, we move on to stride mode and attempt to calculate the possible block size; and we follow a similar validation process before finally falling back on the irregular model if the stride mode cannot be validated.

After detecting the mapping pattern between the child and parent communicator ranks, the next step is to detect the mapping pattern between the child communicator ranks and the AVs/lpids. Recall that the mapping model of a communicator describes how the ranks are translated to AV table indices. Therefore, the mapping model of a child communicator is determined by both the rank map of the parent communicator and the pattern of the mapping between the child and parent communicators. Table II shows the state machine for determining the mapping pattern of the child communicator.

If the child communicator has an irregular mapping model (lut or mlut), the ranks are not necessarily irregularly ordered. The child communicator might have reordered the ranks in the parent communicator to create a regular mapping between the ranks to AV indices. AV-Rankmap performs an additional scan of the ranks to determine whether an irregular model can be converted to a regular mapping model. After all the indirect mappings are processed and the rank map is created, AV-Rankmap scans the lookup table using the same algorithm in step 1 above.

We note that communicator creation is not in the performance-critical path of most applications. Thus, while we do not want to make communicator creation excessively expensive, some additional cost to detect rank-mapping patterns and optimize them is often acceptable. As we will demonstrate in Section V-C, this overhead is between 3-7% in our implementation.

2) Accessing the Rank-Address Translation: In this section, we discuss how the the rank-address translation is accessed in the AV-Rankmap model. This translation is accessed inside the performance-critical path, and hence any overhead created in this path can slow down practically every performance-critical operation in the MPI implementation. So we need to be particularly cautious with respect to the performance overheads of our implementation choices in this path.

The translation is done in three step: 1) checking the mapping model of the communicator; 2) calculating the corresponding index in the AV table of the rank; 3) access the AV table for network address. In order to understand the overhead of our implementation, we use the Intel Software Develop Emulator (SDE) to obtain the instructions that are being executed for the translation. We studied three different implementations for such translation.

The first and most intuitive implementation of the rank-address translation is using a `switch` statement where each case contains the translation code for a specific model. Figure 2 and Figure 3 show the assembly code for translation in the VC-VCRT model and the AV-Rankmap model. For a communicator with *direct* mapping, the AV-Rankmap model uses two additional instructions compared to the VC-VCRT model. Note that for other mapping models like *offset* and *stride*, there are additional instructions for calculating the index from the communicator rank. The use of the `switch` statement introduces four additional instructions (line 1-4 in Figure 3). But we save two instructions for accessing the VCRT (line 1-2 in Figure 2). The difference in the rest of the code is for the addressing changes in accessing the VC/AV table. Note that the `jnbe` instruction in the switch statement is the branch to the default case. This is an unfortunate overhead because the AV-Rankmap model does not have a "default" case and it is not possible to explicitly tell the compiler not to add this branch.

The second alternative implementation that we studied is to use a hybrid `if` branch combined with a `switch` statement. That is, we can simply check if the communicator uses the *direct* mapping model (the `if` branch) and check for other mapping models using a `switch` statement in the `else` branch. This implementation can reduce the cost of translating the *direct* mapping model to eight instruction (as shown in Figure 4). However, the downside of this approach is that all other models in the `switch` statement will have two additional instructions due to the earlier `if` statement. We can further cascade multiple `if` statements to prioritize all the popular models and leave other mapping models in a `switch` statement. But for every level of the cascade, we would add two additional instructions, thus negating the benefit of a direct check after the first level.

In addition to these two models, we can also studied a third implementation that, in essense, manually recreates the branch table that the compiler uses inside a `switch` construct, by using `goto` statements. As one might expect, our initial study (Figure 5) showed that this implementation has the least number of instructions for the branch lookup itself since it has the same two-instruction lookup for all models (similar to `switch`) and allows us to manually disable any extra branches that we do not need (such as the branch to the default case, that the compiler adds for `switch` constructs). However, while integrating this implementation into the overall MPI implementation, we encountered a subtle and unexpected issue. Specifically, when the compiler translates the switch statement

```
1 mov  rax, qword ptr [rbp+0x60]
2 mov  rdx, qword ptr [rip+0x43566d]
3 mov  rax, qword ptr [rax+0x188]
4 mov  eax, dword ptr [rax+r12*4+0xc]
5 shr  eax, 0x1
6 cdqe
7 mov  rax, qword ptr [rdx+rax*8+0x8]
```

Fig. 2: Instruction of Rank-Address Translation in the VC-VCRT Model.

```
1 cmp    dword ptr [rdi+0x1a8], 0xa
2 jnbe   0x435039
3 mov    eax, dword ptr [rdi+0x1a8]
4 jmp    qword ptr [rax*8+0x5cfcc8]
5 movsxd rax, esi
6 add    rax, 0x1
7 shl    rax, 0x4
8 add    rax, qword ptr [rip+0x4682c6]
9 mov    rax, qword ptr [rax]
```

Fig. 3: Instruction of Rank-Address Translation for *direct* Mode in the AV-Rankmap Model.

```
1 mov    edx, dword ptr [rdi+0x1a8]
2 cmp    edx, 0x1
3 jnz    0x435370
4 movsxd rax, esi
5 add    rax, 0x1
6 shl    rax, 0x4
7 add    rax, qword ptr [rip+0x467e8b]
8 mov    rax, qword ptr [rax]
```

Fig. 4: Instruction of Rank-Address Translation for *direct* Mode in the AV-Rankmap Model using "if-switch" Hybrid Implementation.

to `goto` branches, such translation is done *after* the relevant function or macro inlining. Thus, the compiler *sees* all of the inlined functions at the same time and assigns different labels to each `goto` branch. By doing this manually, however, since we do not have information on what functions the compiler might choose to inline, we lose the ability to assign different labels to the various `goto` branches. Consequently, assigning statically decided labels for such branches would, in practice, cause the compiler to disable inlining for such functionality. This causes much more significant overhead compared to the additional branches in the previous two implementations. Thus, we had to abandon this approach and only consider the switch-based and hybrid implementations. We will discuss their performance impact in the Section V.

```
1 mov    eax, dword ptr [rdi+0x1a8]
2 jmp    qword ptr [rax*8+0x5cfbc8]
3 movsxd rax, esi
4 add    rax, 0x1
5 shl    rax, 0x4
6 add    rax, qword ptr [rip+0x47b1c0]
7 mov    rax, qword ptr [rax]
```

Fig. 5: Instruction of Rank-Address Translation for *direct* Mode in the AV-Rankmap Model using "goto" Implementation.

V. Evaluation

In this section, we evaluate the AV-Rankmap model from two perspectives: memory usage and performance.

We used two different test platforms for our evaluation. The first platform is the Mira supercomputer at Argonne National Laboratory, which is a 49,152-node IBM BG/Q system. Each node has 16 cores and 16 GB memory, which allows running 768K processes at the full system scale. Most of our experiments were performed on Mira. However, the BG/Q environment does not provide some capabilities, such as MPI dynamic processes and special tools like Intel Software Development Emulator (which is only available on Intel processors). For experiments that needed these capabilities, we used the Argonne LCRC "Blues" cluster. Each Blues node has two Intel Xeon E5-2670 (8 cores each) and 64 GB memory. We run experiments on Blues up to 256 nodes (4K processes).

The baseline implementation of the evaluation is MPICH 3.2 which uses the VC-VCRT model. The libraries and applications in all experiments are compiled using GCC 4.7.2 with `-O2` option and statically linked.

A. Memory Usage for Communicators with Regular Model

We first focus on the memory usage of AV-Rankmap. For this experiment, we use two microbenchmarks: *split-loop* and *dup-loop*. The *split-loop* benchmark splits the odd and even ranks of `MPI_COMM_WORLD` into two subcommunicators without reordering the ranks. Thus, the ranks in the split communicator have the *stride* mapping model in the AV-Rankmap approach. The *dup-loop* benchmark simply duplicates the `MPI_COMM_WORLD`. Thus, the duplicated communicators would have the *direct* mapping model in the AV-Rankmap approach. As explained in Section II, for every communicator that is created by the user, the MPI implementation internally creates additional convenience communicators: typically a *node* communicator (for ranks on the same node) and a *node-roots* communicator (for root ranks in all nodes).

1) Split Communicators: Figure 9(a) shows the memory usage of 10 and 100 split communicators with up to 768K processes. It is clear that the AV-Rankmap model uses significantly less memory than the VC-VCRT model. At the full scale on Mira, the AV-Rankmap uses only 9 MB of memory for 100 split communicators. The VC-VCRT model, on the other hand, consumes more than 40% of system memory for 10 communicators and exceeds the total system memory for 100 communicators.

Figure 9(b) shows the breakdown of the memory usage for 10 communicators in the AV-Rankmap and the VC-VCRT models with increasing number of processes in the parent communicator. We note that the memory usage of both models is $O(P)$ with respect to the total number of processes in the system: this is because both models need to store the network physical addresses. But, the AV-Rankmap model has a memory usage advantage in two aspects. First, since the size of the network addresses used in AV-Rankmap is smaller (based on the implementation choices described in Section IV-B), the constant associated with the $O(P)$ increase in memory is smaller for this model. In our implementation, we are able to reduce the size of each address vector element (AVE) to 12 bytes in the AV-Rankmap model, which is 40-times smaller than the original 480 bytes in the VC-VCRT model. Second, since the AV-Rankmap model does not use a lookup table in common communicator patterns, but instead dynamically computes the rank to network address translation (based on the implementation choices described in Section IV-C), in the common case we use constant memory instead of an $O(P)$ structure, while the VC-VCRT model uses an $O(P)$ structure

```
1  cmp    dword ptr [rdi+0x1a8], 0xa
2  jnbe   0x435039
3  mov    eax, dword ptr [rdi+0x1a8]
4  jmp    qword ptr [rax*8+0x5cfcc8]
5  movsxd r12, esi
6  mov    eax, dword ptr [rdi+0x1b0]
7  imul   eax, r12d
8  add    eax, dword ptr [rdi+0x1b4]
9  cdqe
10 add    rax, 0x1
11 shl    rax, 0x4
12 add    rax, qword ptr [rip+0x4682c6]
13 mov    rax, qword ptr [rax]
```

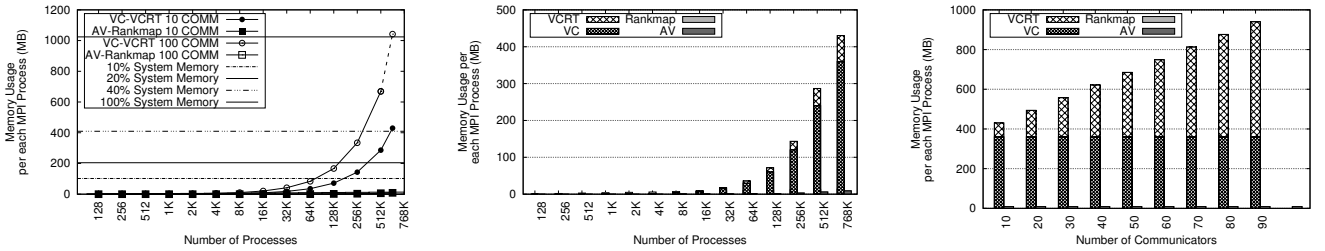
Fig. 6: Instruction of Rank-Address Translation for *stride* Mode in the AV-Rankmap Model.

```
1  mov    edx, dword ptr [rdi+0x1a8]
2  cmp    edx, 0x1 ;if mode == direct
3  jnz    0x435370
4  cmp    edx, 0x3 ;if mode == offset
5  jnz    0x43537
6  cmp    edx, 0x5 ;if mode == stride
7  jnz    0x43538C
8  movsxd r12, esi
9  mov    eax, dword ptr [rdi+0x1b0]
10 imul   eax, r12d
11 add    eax, dword ptr [rdi+0x1b4]
12 cdqe
13 add    rax, 0x1
14 shl    rax, 0x4
15 add    rax, qword ptr [rip+0x467e8b]
16 mov    rax, qword ptr [rax]
```

Fig. 7: Instruction of Rank-Address Translation for *stride* Mode in the AV-Rankmap Model using "if-switch" Hybrid Implementation.

```
1  mov    eax, dword ptr [rdi+0x1a8]
2  jmp    qword ptr [rax*8+0x5cfbc8]
3  movsxd r12, esi
4  mov    eax, dword ptr [rdi+0x1b0]
5  imul   eax, r12d
6  add    eax, dword ptr [rdi+0x1b4]
7  cdqe
8  add    rax, 0x1
9  shl    rax, 0x4
10 add    rax, qword ptr [rip+0x47b1c0]
11 mov    rax, qword ptr [rax]
```

Fig. 8: Instruction of Rank-Address Translation for *direct* Mode in the AV-Rankmap Model using "goto" Implementation.



(a) Different numbers of processes

(b) Detail memory usage of 10 communicators

(c) Detail memory usage with 768K processes

Fig. 9: Memory usage of `MPI_Comm_split` with different numbers of processes and communicators.

for the lookup.

Figure 9(c) shows a different breakdown of the memory usage, this time keeping the number of processes fixed at 768K but increasing the number of communicators created. As expected, the memory usage of the VC-VCRT model grows quickly with the number of communicators because it uses a new $O(P)$ VCRT lookup table for rank mapping on each new communicator. In the AV-Rankmap model, on the other hand, each new communicator uses only a small constant memory space (e.g., to store the offset and stride, which are two integers) if its rank mapping is regular, as is the case in our benchmark.

2) Duplicate Communicators: Figure 10(a) shows the overall memory usage for 10 and 100 duplicate communicators. Unlike the results for the split communicators, where the memory usage of 10 and 100 communicators with the VC-VCRT model are distinctly different, the change in memory usage with duplicate communicators is much smaller. This is because the baseline implementation (MPICH 3.2) already adopts an optimization techniques for duplicate communicators [11]. It allows a duplicated communicator to share the lookup table with its parent communicator. Instead of copying the lookup table, each child communicator only needs to maintain a pointer to the parent's lookup table. However, there is still an increase in memory usage. This is not because of the user-visible communicator itself, but because of the *node* and *node-roots* communicators that the MPI implementation internally creates for each user-visible communicator. These internal communicators are not simple duplicates of the parent communicator since they contain lesser processes and their rank layout is different from that of the parent communicator. This results in these internal communicators creating their own new lookup tables, and thus using additional memory.

In hind-sight, the VC-VCRT model could have used one potential optimization that we had not originally considered. When a new communicator is created (either user-visible or internal), the MPI implementation could search the existing lookup tables that were previously created to see if any of them could be reused for the new communicator. If any of them matched, the MPI implementation could simply reuse the previous lookup table instead of creating a new one. At least for simple duplication of communicators, this approach would have kept the memory usage fixed with increasing number of communicators—the internal *node* and *node-roots* communicators associated with the new user communicator would be able to reuse the VCRTs for the internal *node* and *node-roots* communicators associated with the parent communicator.

We chose to not improve the VC-VCRT model with this optimization for multiple reasons. First, we wanted to keep the model as close to the current state of art as possible. Second, searching all existing lookup tables every time a new communicator is created can be expensive, since each search can take $O(P)$ time unless additional optimization techniques

are used. Third, as shown in Figures 10(b) and 10(c), the VCRT portion of the memory usage is fairly small compared with the VC portion—even at 100 communicators and 768K processes, the VC portion takes nearly 95% of the total memory usage. Thus even with this optimization there would be very small overall gain in memory usage. Fourth, the AV-Rankmap model provides a more generic optimization for all regular communicators instead of the special case of duplicate communicators thus, in some sense, subsuming such optimizations.

3) Regular Intercommunicators: An intercommunicator can also have regular mapping models as long as the ranks in the same group (remote or local) are from the same process group, and the ranks matches one of the regular mapping patterns. We also tested the AV-Rankmap model with regular inter-communicators for completeness. For this experiments, we split the `MPI_COMM_WORLD` into odd and even communicators without reordering, and creates a inter-communicator between the split communicators. Both the remote and local group of the communicators have regular mapping model (stride model). Then, we create split and duplicate the inter-communicator without reordering and measures the memory usages.

Figure 11 shows the memory usage of 10 and 100 split inter-communicators. In this experiment, we perform a odd-/even split on the inter-communicator. Similar to the result of split intra-communicators, the AV-Rankmap model due to the reduction in AVE size and the regular mapping model. Note that, the VC-VCRT model used more memory for the mappings of inter-communicators. This is because, the inter-communicator need to maintain two separate mappings for the remote and local groups. In addition to the *node* and *node-roots* communicators, the inter-communicator also have a *local* communicator for the ranks in the local group. All these three internal communicators will need their own mappings which will be additional lookup tables in the VC-VCRT model. But for the AV-Rankmap model, these additional communicators only need constant memory for each of their mappings because the internal communicators for a regular inter-communicator all have regular mapping patterns.

Figure 12 shows the memory usage of 10 and 100 duplication inter-communicators. In this experiment, we simply duplicate the inter-communicator without reordering. As we expected, the results of this experiments are similar to the duplicate intra-communicator experiments. The AV-Rankmap model is also effective for this use case in reducing the AVE size and exploiting the regular mapping models for the remote group, the local group and the internal communicators. For the VC-VCRT model, the optimization for duplicating communicators has effectively avoided the copying of the lookup table for both the remote and the local groups. It also helped avoiding the copying the lookup table for the *local* communicator. However, as it in the experiment on duplicate intra-communicators, the copying of the lookup tables for the *node* and the *node-roots* communicators are still not avoidable in the VC-VCRT model.

B. Memory Usage for Communicators with Irregular Model

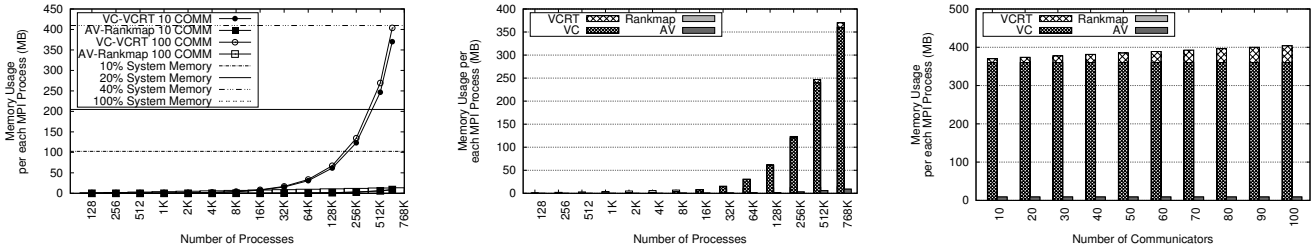
In the AV-Rankmap model, a communicator with irregular rank-to-network-address mapping has to either *lut* model and *mlut* model. These models are usually the results for rank reordering or dynamic process. Here we study the memory usage of these irrgular models.

***lut* with Reordering:** In this experiment, we create a irregular intra-communicator by duplicating `MPI_COMM_WORLD` and reordering the ranks. We ensure that each duplicate communicator has a unique order of ranks. This prevents any reuse of the lookup table between communicators. In this worst case scenario, both the AV-Rankmap model and the VC-VCRT model have to allocate one lookup table for each communicator, i.e. $O(p)$ memory for mapping. Figure 13 shows the memory usage of 10 and 100 dup-reordered communicators. Both the AV-Rankmap model and the VC-VCRT model consumed a large amount of memory due to the lookup tables. Note that, the element size of the lookup table of these two models are the same. Thus, the memory usage for mapping in both models grows at the same pace. However, the AV-Rankmap model still has lower memory usage than the VC-VCRT model due to the fact that the AVE in the AV-Rankmap model is smaller than the VC in the VC-VCRT model.

Communicators with Dynamic Process: One "optimization" that we used in the AV-Rankmap model for VC compression is to move the process group ID from the VC to the communicator. The reasoning behind this optimization was that since most applications do not use dynamic processes, they should not be penalized with the memory overhead associated with maintaining the associated metadata. Only applications that create communicators with dynamic processes must face this overhead. This helps us reduce the size of the AVE in the AV-Rankmap model, as demonstrated above. The flip-side of this optimization, however, is that applications that do use dynamic processes are penalized more heavily since each new communicator that is not a simple remapping of the parent communicator ranks would now need to maintain the expensive *mlut* lookup table.

Put another way, in the VC-VCRT model the memory overhead of maintaining the process group ID is attached to each VC. Thus, it would increase the base memory usage but would not add extra memory usage for each new communicator created. In the AV-Rankmap model, the memory overhead of maintaining the process group ID is attached to each communicator. Thus, the base memory usage would be small, but if the ranks of the new communicator are arbitrarily reordered compared to the parent communicator, the incremental memory usage per communicator would be $O(P)$ where P is the size of the communicator.

Here we present our evaluation for the case where a single dynamic-process communicator is created. We used the LCRC Blues system, which is an InfiniBand-based linux cluster, for this experiment because the Mira supercomputer does not support spawning dynamic process. We create a inter-communicator with dynamic process and merge it into a intra-communicator. The intra-communicator is *mlut* mapping model. We then duplicate this intra-communicator and reorder



(a) Different numbers of processes (b) Detail memory usage of 10 communicators (c) Detail memory usage of 768K processes

Fig. 10: Memory usage of `MPI_Comm_dup` with different numbers of processes and communicators.

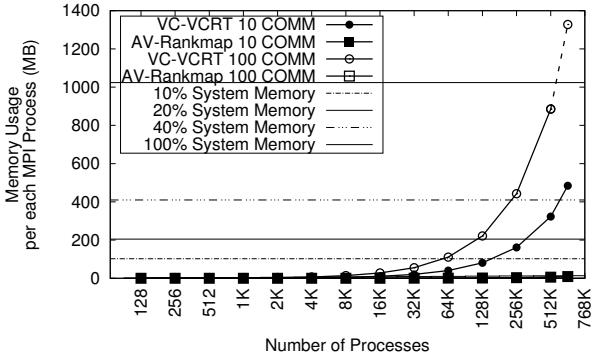


Fig. 11: Memory Usage of 10 and 100 Split Inter-Communicators.

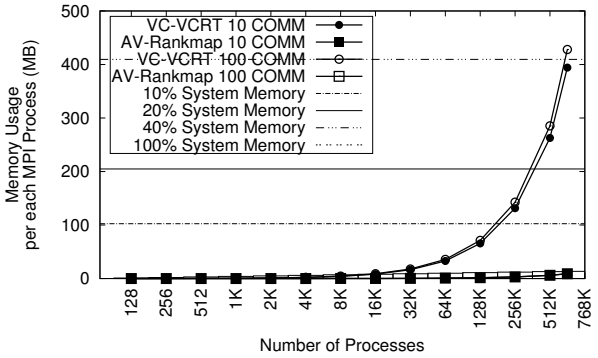


Fig. 12: Memory Usage of 10 and 100 Duplicate Inter-Communicators.

than ranks in the new communicator. The rank order in each new communicator is unique to prevent lookup table sharing. Figure 14(a) shows the memory usage for 10 and 100 dup-reordered communicators in the VC-VCRT and AV-Rankmap models. While AV-Rankmap model used less memory than the VC-VCRT model for 10 communicators, it used significantly more memory than VC-VCRT for 100 communicators. As Figure 14(b) shows, the AV-Rankmap model uses about twice the amount of memory to maintain the associated lookup table. This is because each element in the lookup table in AV-Rankmap stores both the AV table ID and the AV table index (*mlut* model), while the lookup table in the VC-VCRT model stores only the VC index. The trend of memory usage is more clear when we fix the number of processes to 4096 and change the number of communicators. As shown in Figure 14(c), the memory usage of the AV-Rankmap grows rapidly as we increase the number of communicators.

While this memory overhead is certainly an issue with the design choice made in the AV-Rankmap model, we believe this tradeoff is worthwhile for most applications since dynamic processes are rarely used in large MPI applications. In fact, many systems (like Blue Gene and Cray) do not even support such capabilities. We further note that for other communicators derived from the *mlut*-based communicator, the previously discussed mapping compression techniques still hold. For example, duplicate communicators created from such a communicator would not create another *mlut* table and would use only constant memory space.

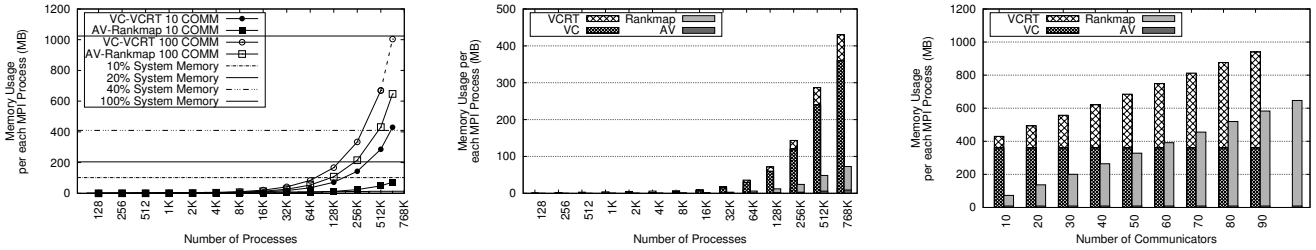
C. Performance

There are two performance aspects that need to be studied: (1) communicator creation overhead and (2) network address lookup overhead. As discussed earlier, communicator creation is not on the performance-critical path for most applications. Thus, while we do not want to make it too expensive, some overhead is typically acceptable. Network address lookup, on the other hand, must not create any additional overhead for the approach to be practically viable.

Communicator Creation Cost: In the AV-Rankmap model, when a new communicator is created the MPI implementation checks to see if the rank mapping of the new communicator matches any of the regular patterns that it understands. If it does match a regular pattern, it stores the associated simple metadata (e.g., offset and stride) instead of creating a full lookup table for the network address translation. This check for regular patterns, however, adds some overhead to the communicator-creation path.

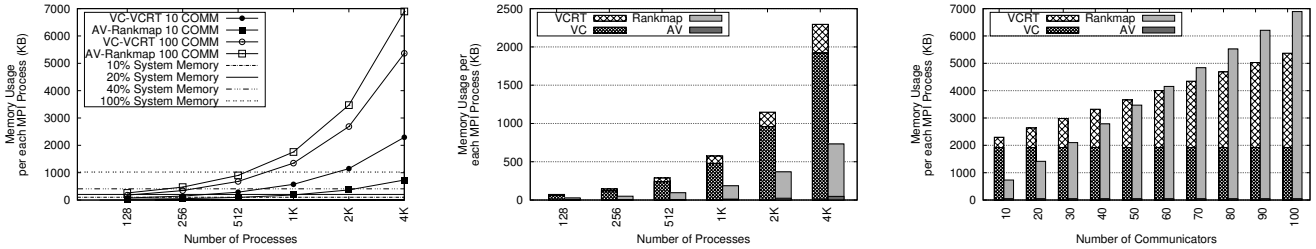
We measure the communicator creation overhead using Mira and LCRC Blues. The reason for this setup is that Mira has no support for dynamic process. The inter-communicator creation on Mira can only be done with ranks in the same `MPI_COMM_WORLD`. For completeness, we run the same experiments on LCRC Blues with dynamic process. Figure 15 and Figure 16 shows the overhead on communicator creation time on Mira and LCRC Blues, respectively.

Overall, the AV-Rankmap model has 3%–6% overhead in communicator creation on Mira. For `MPI_Comm_dup`



(a) Different numbers of processes (b) Detail memory usage of 10 ireregular communicators (c) Detail memory usage of 768K processes

Fig. 13: Memory usage of irregular communicators with different numbers of processes.



(a) Different numbers of processes (b) Detail memory usage of 10 ireregular communicators (c) Detail memory usage of 768K processes

Fig. 14: Memory usage of mlut communicators with different numbers of processes.

and `MPI_Comm_split`, the overhead is less than 5% at full scale of Mira (768K processes). The overhead is due to the detection of mapping patterns. The overhead for `MPI_Intercomm_create` on Mira is also less than 5%, while the overhead on LCRC Blues is 6%–8%. When running on Mira, the exchange of network address between the local and remote groups during the inter-communicator creation is bypassed because all the processes are from the same `MPI_COMM_WORLD`. This saved time for identifying new process groups, creating new AV table and inserting the network addresses, which in turn compensated the cost of detecting mapping model. `MPI_Intercomm_merge` also have lower overhead on Mira than it on LCRC Blues for similar reason. The major overhead of merging a inter-communicator is merging the mapping of the remote and local groups. This merging is likely to create a new mapping with irregular model. Therefore, our implementation will try to allocate *mlut* for the compatibility for arbitrary mapping with dynamic process. Later, it will try to detect the mapping patterns and reduce the *mlut* to a *lut* or a regular model. However, when running on Mira, our implementation can detect that the remote and local groups only contains the ranks from the same `MPI_COMM_WORLD`, which allows it to bypass the creation of *mlut*.

Rank-to-Network-Address Lookup Performance: As we introduced in Section IV-C2, in order to lookup the network address corresponding to a communicator rank, the MPI implementation must first lookup what mapping model that communicator is using and then using that information either compute or lookup the actual network address. For this experiment, we

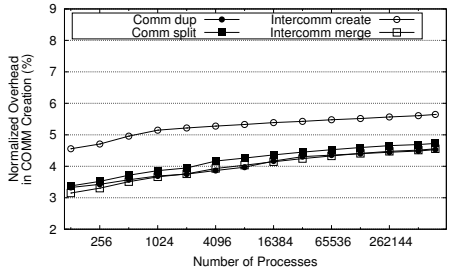


Fig. 15: Creation time for different communicators on Mira.

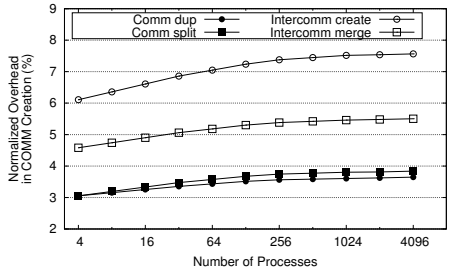


Fig. 16: Creation time for different communicators on Blues.

developed a microbenchmark that issues `MPI_Put` operations to each rank in a communicator in a round-robin fashion. The message size is 8 bytes (a *double*). The microbenchmark set to send one million `MPI_Put` messages to each rank;

We perform the experiment with communicators of five mapping models: *direct*, *offset*, *stride*, *lut* and *mlut*. All communicators used in the experiments has half of the ranks in `MPI_COMM_WORLD`. We use the low/high spliting to create the *direct* and the *offset* communicators. We use the odd/even

spliting to create the *split* communicator. We reverse the order of the ranks in the *split* communicator to create the *lut* communicator. The *mlut* communicator is created by merging a inter-communicators with 4 local processes and 4 spawned processes.

We measure the instruction count and the cache misses for the rank-to-network-address lookup operation. We also measure the issuing rate of `MPI_Put`. We study the issuing rate from two aspects: (1) the theoretical maximum issuing rate and (2) the issuing rate on real network hardwares. To measure the theoretical maximum issuing rate, we bypass the low level network transport. In such a way, each `MPI_Put` call will go through the entire software stack of the MPI library and return right before the data is being sent out. The issuing rate is bounded the by the cost of the MPI library and the processing speed of the CPU (including frequency and number of integer units in the pipeline). Any overhead in terms number of instructions or cache misses can potentially have a huge impact on the issuing rate. The issuing rate on real network hardwares is measured through running the microbenchmark with unmodified MPI libraries on Mira and Blues. Mira uses IBM's proprietary 5D-torus network and Blues uses QLogin Infiniband QDR network.

Instruction-count Analysis: Table III shows the instruction count for rank-to-network-address translation using the `switch`-based implementation, the `if-switch` hybrid implementation and the `goto` statement. The baseline implementation uses 7 instructions for rank-to-network-address translation. As we introduced in Section IV-C2, the `switch`-based implementation in *offset* model uses two more instructions than the baseline. It adds four instructions for the `switch` statement and saves two instruction from access the lookup table. The `switch`-based implementation in *offset* model uses four more instructions than the baseline. On top of the *direct* model, it adds two instructions for loading and adding the offset value.to the rank. On top of the *offset* model, the *stride* model adds another two instrictions for calculating the stride. The *lut* model adds spend two instructions to access the lookup table. The *mlut* model further need three more instructions on top of the *lut* model. One of them is for loading the AV table ID, and the other two are for loading the address of the given AV table ID.

For the *direct* model, using the hybrid implementation saves the branch to the default case which is not needed in our implementation. However, it adds overhead to other models. For example, the *offset* model in the hybrid implementations needs four more instructions that the *direct*. While two of them are for offset calculation, the other two for the `if` statement that checks for *offset* model. The overhead of the instructions for these `if` statement is carried to other models which makes the instruction count for the *stride*, *lut* and *mlut* to be a lot higher than them in the `switch` implementation.

We have also show the instruction count for the `goto`-based implementation. As we explained in Section IV-C2, compiler cannot inline the rank-address translation function because the usage of addresses to the global labels. Although it has the minimum number instructions for *translation*, the function call costs 15 instruction. Such a high instruction count makes the `goto`-based implementation less efficient.

TABLE III: Instruction Count for Rank-to-Network-Address Translation for Different Mapping models. The baseline VC-VCRT model uses 7 instructions.

	direct	offset	stride	lut	mlut
switch	9	11	13	11	15
hybrid	8	12	16	18	21
goto	22	24	26	24	28

Cache Analysis: We mentioned that the performance improvement of the AV-Rankmap model is mainly due to the better cache locality for the shrinked VC. We use the same microbenchmark as in the issuing rate experiment and instrument the code using PAPI (on Blues) and BGPM (on Mira) to measure the cache misses. The cache has been warmed up before measuring. The experiment is done on one Blues node with 16 cores. Each process issues 8 million `MPI_Put` operations. Table IV and Table V show the cache misses for each level of cache on Blues and Mira.The results show that when running with the VC-VCRT model (baseline), the test program experienced high misses in L1D cache and L2 cache. The VC-VCRT model uses large VC objects (480B each) to store information such as network address of process and function pointers to the transport functions which is used during the communication. As we discussed in Section IV-B, the AV-Rankmap model shrinks the size of VC by moving these information to separate structures, which effectively reduces the cache misses due to accessing different cache lines of the VC object. At the same, the access to common structures is also more cache friendly. For example, All processes usually share the same set of the network transports, storing the function pointers in a dedicated structure improved the cache locality.

Message Issue Rate: Table VI shows the issuing rate with communicators with regular mapping patterns. Overall, the AV-Rankmap model outperformed the baseline in all scenarios and achieved up to 50% higher issuing rate than the baseline. The performance improvement is mainly because the smaller VC size leads to better cache locality for VC objects. We will discuss this in detail later.

The difference in issuing rate between different mapping models reflects the extra work in rank translation in these models. As we mentioned in Section IV-C2, the *direct* model is expected to have the highest issuing rate because there is no extra calculation or lookup in translation. The *offset* model is slightly slower than *direct* due to the addition of the offset value. The *stride* is even slower for the extra calculation for the stride value. Although the *lut* model does not have extra calculation, accessing the lookup table can potentially be slower than the others depending the cache locality of the lookup table. The results in Table VI generally confirms these predictions. That is the *direct* model has the highest issuing rate while the *offset* model and the *stride* model has slightly lower issuing rate. As we mentioned above the theoretical maximum issuing rate is bounded by the CPU performance. The differences between Mira and Blues is mainly because

TABLE IV: Cache Misses for one Process Issuing 8 Million `MPI_Put` on Blues.

Cache	Baseline	direct	offset	stride	lut	mlut
L1D	120491	103	168	141	98	192
L2	237	47	43	69	53	60
L3	47	48	47	46	47	45

TABLE V: Cache Misses for one Process Issuing 8 Million `MPI_Put` on Mira.

Cache	Baseline	direct	offset	stride	lut
L1D	180491	184	191	189	178
L2	422	57	53	61	68

Mira's CPU has lower frequency (1.6 GHz vs 2.7 GHz on Blues) and less integer units (1 per core vs 4 per core on Blues). Note that the placement of generated instructions can also affect the performance. One example is for the switch-based implementation on Blues. The *offset* model and the *stride* model share the instruction for adding offset to the translated rank. This introduces one extra branch instruction from jumping from the code path of *offset* to the code path of *stride*. This is why the *offset* model has lower performance than the *stride* model in the experiment. Unfortunately, the compiler does not provide a way to control the code placement of a switch statement. Thus there is no guarantee on the branch overhead of each model in switch-based implementation.

The only way to control the branch overhead between different models is using the hybrid implementation. As we explained earlier, it uses multiple cascade if statement to control the checking order of the models. It is expected to be faster than the switch-based implementation in *direct*, because of it avoids the relative expensive table jump and saves of the branch to the default case in the switch. However, in the experiment, the branch table of the switch statement was mostly in cache, thus reducing the overhead of table jump and diminished the benefit of the hybrid model. On the other hand, the other mapping models in the hybrid implementation has to pay the cost of extra branches due to the cascaded if statement which led to a much worse performance than the switch-based implementation. Base one these observations, we decide to use switch-based implementation for the rest evaluations.

Table VII shows the `MPI_Put` issuing rate through real network hardware on Blues and Mira. The the AV-Rankmap model and the VC-VCRT model has almost the same issuing rate on both machines and with all different communicators. Due to the cost of transmitting data over the network, the performance difference that we mentioned above does not demonstrate its impact on the communication performance. However, the native hardware support for MPI operations and low latency nature of future high-performance network already poses the need of a lightweight MPI library implementation. Improving the performance can potentially benefit future systems.

D. Applications

Here we evaluate AV-Rankmap with the miniapps presented in Section III. Because of limited space, we show the results only for *BT*, *FT*, and *SP*, from the NAS parallel benchmarks [4]; *AMG2013*, *Nekbone*, from the CORAL benchmarks [3]; and *AMR*, from the ExACT codesign center [2]. All these miniapps were run on Mira. The experiments scaled from 16K processes to 512K processes.

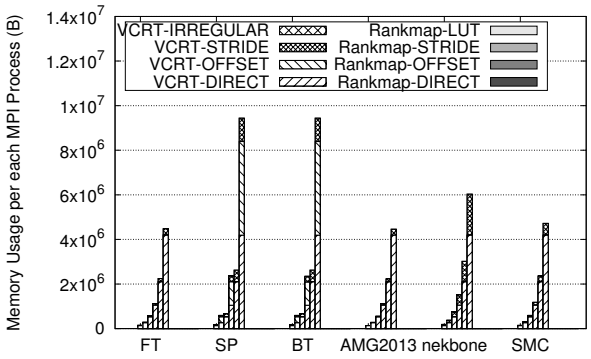


Fig. 17: Memory usage of rank-process mapping structure of miniapps on 16K to 512K processes.

Overall, the miniapps create 2–7 user communicators. Therefore, the size of all the VC objects is still the dominating factor. The memory usage of these miniapps on 16K processes is around 15 MB and linearly increases to 244 MB (25% of the system memory) on 512K processes. In contrast, AV-Rankmap uses only 128 KB to 4 MB of memory on 16K processes to 512K processes. These miniapps employ different ways to create communicators, thus leading to different memory usage patterns for the VCRTs as shown in Figure 17. We divide the memory usage of the VCRTs based on the process mapping models of communicators.

The VCRT for *direct* mapping model communicators uses a significant amount of memory, mainly because of the VCRT allocated for `MPI_COMM_WORLD`. Since the duplication of `MPI_COMM_WORLD` makes only a shallow copy of the parent's VCRT, *Nekbone* (which creates 7 dups of `MPI_COMM_WORLD`) uses almost the same amount of memory as does *AMG2013* (which creates only one dup of `MPI_COMM_WORLD`). As mentioned in Section II, MPI creates local communicators (node_comm and node_root_comm) with `MPI_COMM_WORLD`. These communicators also get duplicated with `MPI_COMM_WORLD`. VCRTs for offset and stride communicators in *Nekbone*, *AMG2013*, and *SMC* all are used for these local communicators.

Besides the use of *MPI_Comm_dup*, *BT*, *SP*, and *FT* create communicators can also be created by using `MPI_Comm_split`. *BT* and *SP* actually create one large communicator with the offset model. These two benchmarks prefer the number of processes to be a square number. When a nonsquare number of processes such as 512K or 128K is used, they will try to reduce the number of processes to the closet square number. They use `MPI_Comm_split` to create a offset communicator that is almost the size as

TABLE VI: Theoretical maximum issuing rate per each MPI process on Mira and Blues with *switch* and *hybrid* rank-address translation.

	Baseline	DIRECT	OFFSET	STRIDE	LUT	MLUT
Mira (switch)	11088082	16633243	15486542	15081816	16005225	N/A
Mira (hybrid)	11274682	16641417	16667171	14465111	14935122	N/A
Blues (switch)	64762341	98799192	92134284	96640891	96031355	84512392
Blues (hybrid)	64762341	99848504	93003065	85790667	89610737	82112870

TABLE VII: Issuing rate per each MPI process using real network on Mira and Blues with *switch* and *hybrid* rank-address translation.

	Baseline	DIRECT	OFFSET	STRIDE	LUT	MLUT
Mira (switch)	899888	898876	899381	896354	896860	N/A
Mira (hybrid)	899888	898741	8988278	896162	898547	N/A
Blues (switch)	4324721	4338395	4323876	4311984	4324431	4316782
Blues (hybrid)	4324721	4327688	4330049	4321378	4314790	4310411

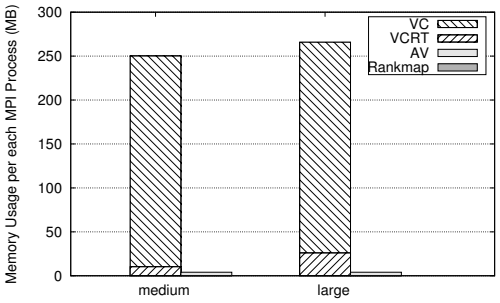


Fig. 18: Memory consumption of process address translation of Nek5000 on 512K processes.

`MPI_COMM_WORLD`. *FT* organizes the processes in a two-dimensional grid and creates "row" communicator and "column" communicators. The size of the "row" and "column" depends on the number of processes and dimension of the problem. Since all the communicators have a regular mapping model, the memory usage of rank maps in AV-Rankmap is small: 1.3 KB for *Nekbone*, which has the most communicators in these miniapps.

We also tested AV-Rankmap with the full applications mentioned in III. Because of space limits, however, we present the results only for *Nek5000*. We evaluate AV-Rankmap with two Nek5000 cases. The first case is a medium-size problem that uses the XXT solver, which creates 24 duplicates of `MPI_COMM_WORLD`. The second case is a large problem that uses the highly scalable AMG solver, which creates 86 duplicates of `MPI_COMM_WORLD`. We ran both cases with 512K processes (32,768 nodes, 16 processes per node) on Mira and measured their memory usage of the communicator.

Figure 18 presents the per-process memory consumption for communicators. On 512K processes, the VCs and VCRTs consume a large amount of memory, whereas the AV and rank maps use very little memory. The memory size for VCs in both cases is the same because of the identical number of processes. The increase in memory consumption for the VCRT-VC scheme in the large Nek5000 case is due to the growth in the VCRT memory. The large Nek5000 case creates more communicators than does the medium-size case, thus leading to higher memory usage on VCRTs.

VI. Related Work

Balaji et al. [8] discuss the memory overheads of communicators in MPI and note that memory usage increase with the number of processes significantly affects the number of communicators that can be created. They report that on Blue Gene/P the number of new communicators that can be created on 128K processes drops to as low as 264 from 8,189. Their findings strengthen the argument for compressing the memory usage of process address translation in communicators.

Several other works focus on reducing the memory usage of MPI communicators and groups [9], [17]. In order to support the various possible group patterns, these approaches have complex models for saving the ranks in groups. However, the approaches incur high overhead in rank-address translation. For example, the sparse group proposed in [9] has been adopted as an optional feature in OpenMPI. The rank to network address translation is from a child communicator to a parent communicator, rather than directly from a child communicator to the actual network address. This means that as more communicators are created, the translation needs to iteratively traverse the tree of the ancestor communicators to get to the actual network address, thus significantly increasing the number of instructions required to do the translation. We performed a case study on the performance of the sparse groups approach. We first perform an odd/even split on the `MPI_COMM_WORLD` to create the first generation of split communicator. Then we derived the next generation of split communicator by split the current generation in the same odd/even manner. Therefore, all split communicators has stride mapping model. Figure 19 shows the theoretical message issuing rate on different generation of split communicators using the default OpenMPI, OpenMPI with sparse groups and AV-Rankmap. The objective of this case study is to show the trend of performance for each approach on different generations of split communicators. Due to the iterative traversing of the tree of the ancestor communicators, the OpenMPI with sparse group has around 10% performance loss for each additional generation of communicators. On the other hand, the performance of both the default OpenMPI and AV-Rankmap is invariant to the depth of the tree of ancestor communicators. Also, due to the fundamental differences in the implementation, the OpenMPI and MPICH has a significant difference in performance in

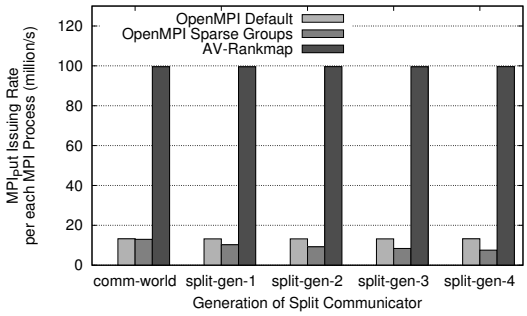


Fig. 19: Theoretical `MPI_Put` Issuing Rate for Different Generations of Split Communicators.

the case study. Besides the performance difference, the sparse groups in OpenMPI still requires MPI_COMM_WORLD to have an O(P)-size lookup table, where P is the number of processes created.

Some studies propose approaches to distribute the table for ranks among multiple processes [12], [13], [15]. Sack and Gropp [15] propose a distributed algorithm for ordered child communicator construction that uses $O(n/p)$ memory by distributed tables for storing the ranks. Recent work by Moody et al. [14] mentions a generalized MPI_Comm_split. They propose creating and storing process groups as chains in $O(1)$ memory and $O(\log n)$ construction time. They perform collectives by exchanging appropriate process ids during the operation. As we demonstrated in our case study, however, the rank table is not necessary for most communicators. Also, because of the nature of the distributed rank table, some rank-process translations need additional communication, which adds a large overhead to the performance.

A more successful communicator memory compression technique used by MPICH is proposed by Goodell et al. [11]. This approach allows duplicated communicators to share the same VCRT as their parent, which removes multiple copies of the same VCRT. As we mentioned earlier in the paper, however, this approach eliminates the need for VCRTs only in limited cases. Even for duplicate communicators, internal communicators used in MPI are not direct duplicates and thus cannot use the same approach. Nevertheless, this approach is widely used in many MPI implementations and is, in fact, the baseline case that we compare against in this paper.

Compared with all these previous studies, AV-Rankmap has three major differences: it eliminates the need for a rank table for the majority of use cases of child communicators; it uses a simple process mapping model that avoids the overhead of complex mapping techniques and distributed mapping tables; and it tackles compression in both the VC structure and the VCRT mapping model.

VII. Conclusion

We proposed a new mechanism, called AV-Rankmap, for network address management in MPI. AV-Rankmap detects patterns in rank mapping that applications naturally tend to have, as well as the fact that some aspects of the network address translation are naturally more performance critical than others. It uses this information to compress the network address management structures. We demonstrated that AV-Rankmap significantly reduces the memory usage of communicators.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Blues , a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] Center for Exascale Simulation of Advanced Reactors. https://cesar.mcs.anl.gov.

[2] Center for Exascale Simulation of Combustion in Turbulence. https://cesar.mcs.anl.gov.

[3] CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks.

[4] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[5] Nek5000. https://nek5000.mcs.anl.gov.

[6] QBOX. http://computation.llnl.gov/projects/qbox-computing-structures-quantum-level.

[7] SNAP: SN Application Proxu. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/snap/.

[8] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a Million Processors. *Parallel Processing Letters*, 21:45–60, 2011.

[9] Mohamad Chaarawi and Edgar Gabriel. *Computational Science – ICCS 2008: 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I*, chapter Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators, pages 297–306. 2008.

[10] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, chapter Noncollective Communicator Creation in MPI, pages 282–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[11] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable Memory Use in MPI: A Case Study with MPICH2. In *Proceedings of the 18th EuroMPI Conference (EuroMPI)*, 2011.

[12] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of Communicators and Groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[13] Humaira Kamal and Alan Wagner. An Integrated Fine-Grain Runtime System for MPI. *Computing*, 96(4):293–309, 2014.

[14] Adam Moody, Dong H. Ahn, and Bronis R. de Supinski. Exascale Algorithms for Generalized MPI_Comm_Split. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, 2011.

[15] Paul Sack and William Gropp. A Scalable MPI_Comm_Split Algorithm for Exascale Computing. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, 2010.

[16] M. Si, A. J. Pea, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 665–676, 2015.

[17] Jesper Larsson Träff. *Recent Advances in the Message Passing Interface: 17th European MPI Users' Group Meeting, EuroMPI 2012, Stuttgart, Germany, September 12-15, 2012. Proceedings*, chapter Compact and Efficient Implementation of the MPI Group Operations, pages 170–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[18] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations . *Computer Physics Communications*, 181(9):1477–1489, 2010.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.